STUDIA UNIV. "BABEŞ–BOLYAI", INFORMATICA, Volume **XLIV**, Number 1, 1999

PARALLEL PROGRAMS DESCRIPTION WITH *POWERLIST*, *PARLIST* AND *PLIST*

VIRGINIA NICULESCU

Abstract. Data structures *PowerList*, *ParList* and *PList* are efficient tools for functional descriptions of parallel programs, that are *divide and conquer* in nature. J. Misra and J. Kornerup have introduced these theories. This paper presents these theories by suggestive examples, taken from numerical analysis. A case study - Fast Fourier Transformation - illustrates the power of this method for correct parallel programs developing.

1. Introduction

PowerList, *ParList* and *PList* are data structures that can be successful used in a simple functional description of parallel programs, that are *divide and conquer* in nature. They allow working at a high level of abstraction, especially because the index notations are not used. To assures methods to verify the correctness of the parallel programs, algebras and induction principles are defined on these data structures.

1.1. **Notations and Types.** The following basic data types are used: natural numbers ($Nat(\geq 0)$), positive natural numbers ($Pos(> 0)$), real numbers (*Real*), complex numbers (*Com*), and booleans (*Bool*). These types are called scalar types. A type is generally denoted by X. The type of a function is specified by giving the name of the function, its domain and its range.

Function application is written by an infix, left associative, dot operator ($f.x$).

2. *PowerList*

A *PowerList* is a linear data structure whose elements are all of the same type. The length of a *PowerList* data structure is a power of two. The type constructor for *PowerList* is:

$$PowerList : Type \times Nat \rightarrow Type$$

and so a *PowerList* with 2^n elements of type X is specified by $PowerList.X.n$. A *PowerList* with a single element a is called *singleton*, and is denoted by $< a >$. If two *PowerList* structures have the same length and elements of the same type, they are called *similar*s.

Two *similar PowerLists* can be combined into a *PowerList* data structure with double length, in two different ways:

- using *tie* operator $p \mid q$; the result contains elements from p followed by elements from q
- using *zip* operator $p \natural q$; the result contains elements from p and q, alternatively taken.

Therefore, the constructor operators for *PowerList* are:

$$\begin{array}{rcl} < . > & : & X \to PowerList.X.0 \\ .|. & : & PowerList.X.n \times PowerList.X.n \to PowerList.X.(n+1) \\ .\natural. & : & PowerList.X.n \times PowerList.X.n \to PowerList.X.(n+1). \end{array}$$

PowerList algebra is defined by these operators and by axioms that assure the existence of unique decomposition of a *PowerList*, using one of *tie* or *zip* operator; and the fact that *tie* and *zip* operators commute [1].

On *PowerList* data structures an induction principle is defined, that allows function definitions, and the proving of *PowerList* properties. If $\Pi : PowerList.X.n \to Bool$ is a predicate, the induction principle is:

$$\begin{array}{l} ((\forall x : x \in X : \Pi. < x >) \\ \quad \wedge((\forall p, q, n : p, q \in PowerList.X.n \wedge n \in Nat : \\ \qquad \Pi.p \wedge \Pi.q \Rightarrow \Pi.(p \mid q)) \\ \quad \vee(\forall p, q, n : p, q \in PowerList.X.n \wedge n \in Nat : \\ \qquad \Pi.p \wedge \Pi.q \Rightarrow \Pi.(p \natural q))) \\ \Rightarrow \\ \quad (\forall p, n : p \in PowerList.X.n \wedge n \in Nat : \Pi.p). \end{array}$$

Reduction function is an example of higher order function, a function that takes a function as an argument:

$$reduce : (X \times X \to X) \times PowerList.X.n \to X$$

The first argument is an associative binary operator on X type. The *reduce* function is defined by:

$$\begin{array}{rcl} reduce.\odot.< a > & = & a \\ reduce.\odot.(p \mid q) & = & reduce.\odot.p \ \odot \ reduce.\odot.q. \end{array}$$

The *sum* function is an example of a reduction. It calculates the sum of all elements of a *PowerList*, and $sum = reduce.(+)$.

Numerical Integration Using Recursive Romberg Formula

Let $f : [a, b] \to \Re$ be a function. The integral $I = \int_a^b f.xdx$ can be approximate with Romberg formula by [4]:

$$Q_{T_k}.f = \frac{1}{2} Q_{T_{k-1}}.f + \frac{h}{2^k} \sum_{j=1}^{2^{k-1}} f.(a + \frac{2j-1}{2^k} h),$$

where $h = b - a$, $k = 1, 2, \ldots$.

When k converge to ∞, the sequence $(Q_{T_k}.f)$ converge to value I.

For a fix k, a division on $[a, b]$ interval is taken:

$$[x_0, \ldots, x_k] \quad = \quad [a, a + \tfrac{h}{2^k}, \ldots, a + \tfrac{2^k-1}{2^k}h].$$

We define a function *Rom* on *PowerList*, that calculate the $(Q_{T_k}.f)$ value, for a fix k. This function has as argument the *PowerList* of the values of function f on the division points: $p = [f.x_0, \ldots, f.x_k]$. It can be noticed that the even positions elements intervene in the second term of the sum that give the value of $(Q_{T_k}.f)$. Therefore, in the structural definition of function *Rom* will be used the *zip* operator.

$$Rom : Real \times Real \times PowerList.Real.k \rightarrow Real.$$

The first argument, $hk = \frac{h}{2^k}$, is the division step, the second argument is the value of function f in b, and the third is the *PowerList* that contains function values.

$$\begin{array}{lcl} Rom.hk.fb. < x > & = & \frac{1}{2} * hk * x + \frac{1}{2} * hk * fb \\ Rom.hk.fb.(p \natural q) & = & \frac{1}{2} * Rom.(2 * hk).fb.p + hk * sum.q \end{array}$$

To prove the correctness of this algorithm the predicate $\Pi.p \equiv Rom.hk.fb.p = Q_{T_k}.f$ is chosen.

Some observations can be made about *PowerList* notation.

- The parallelism is implicitly introduced by constructors operators, *tie* and *zip*.
- The correctness for an algorithm can be proven using the induction principle.
- The *PowerList* function can be efficiently implemented on hypercube architectures [2]. If we label each element of a *PowerList* of length 2^n with a bit string (of length n), representing the position of the element in the *PowerList*, this element can be mapped to the node with the same label on a hypercube of size 2^n. By the construction above, it follows by induction that the *zip*(*tie*) of the representation of two *PowerList* can be implemented efficiently by combining the representing cubes in the low (high) order bit.

3. ParList

The *ParList* data structure is analogue with *PowerList*, with the difference that the number of the elements is not a power of two.

The type constructor for *ParList* is:

$$ParList : Type \times Pos \rightarrow Type$$

and a *ParList* with n elements of type X is specified by $ParList.X.n$.

It is necessary to use other two operators: *cons*($\triangleright$) and *snoc*($\triangleleft$); they allow to add an element to a *ParList*, at the beginning or at the end of the *ParList*.

The constructor operators are:

$$\begin{array}{rcl} <.> & : & X \to ParList.X.1 \\ .\triangleright. & : & X \times ParList.X.n \to ParList.X.(n+1) \\ .\triangleleft. & : & ParList.X.n \times X \to ParList.X.(n+1) \\ .|. & : & ParList.X.n \times ParList.X.n \to ParList.X.(2n) \\ .\natural. & : & ParList.X.n \times ParList.X.n \to ParList.X.(2n) \end{array}$$

Axioms of *ParList* algebra express like those from *PowerList* algebra, the existence of a unique decomposition of *ParList*, using constructor operators, the commutativity of *tie* and *zip* operators, and some axioms that make connection among operators [1].

An induction principle is also defined for *ParList*. But, in this case, a proving has three stages: the base case, the odd inductive stage and the even inductive stage. The rule of structural decomposition for *ParList* is as follows: when the number of the elements is even is used *tie* or *zip* operators, and when this number is odd is used *cons* or *snoc*. This way, the decomposition is unique.

The *ParList* function definition must contains definitions corresponding to these three stages.

For example, the *map* function

$$map : (X \to Y) \times ParList.X.n \to ParList.Y.n$$

is defined by:

$$\begin{array}{lcl} map.f.<a> & = & <f.a> \\ map.f.(p|q) & = & map.f.p \mid map.f.q \\ map.f.(a \triangleright q) & = & f.a \triangleright map.f.q. \end{array}$$

where the first argument is a scalar function on X type.

Operators on type X can be extended over *ParList.X* in the following way. Let $\odot$ be a binary associative operator on type X, $\odot : X \times X \to X$.

The extended operator:

$$\odot : ParList.X.n \times ParList.X.n \to ParList.X.n$$

is defined by:

$$\begin{array}{lcl} <a> \odot <b> & = & <a \odot b> \\ (p \mid q) \odot (u \mid v) & = & (p \odot u) \mid (q \odot v) \\ (a \triangleright p) \odot (b \triangleright q) & = & a \odot b) \triangleright (p \odot q). \end{array}$$

It is possible to consider another inductive principle, which excludes *tie* and *zip* operators. Even if, using such a principle seems to not lead to a good parallelisation, there are cases for which the parallelism is quite appreciably.

Divided Differences

Let $X = \{x_0, \ldots, x_m\}$ be a set, and $f : X \to \Re$ a function. The kth order divided difference of function f in x_r is [4]:

$$(D^k f).x_r = \frac{(D^k f).x_{r+1} - (D^k f).x_r}{x_{r+k} - xr}$$

where $r, k \in Nat, r < m \wedge 1 \leq k \leq m - r$.

We consider two *ParList* data structures p and q, with elements the division points, and the values of function f in the division points:

$$p = [x_0,\ \ldots,\ x_m] \quad \text{and} \quad q = [f.x_0,\ \ldots,\ f.x_m]$$

The function

$$dif : ParList.Real.n \times ParList.Real.n \to Real$$

that calculate $D^{m-1}f$, is defined by:

$$\begin{array}{lcl} dif.[a\ b].[x\ y] & = & (y - x)/(b - a) \\ dif.(a \triangleright p \triangleleft b).(x \triangleright q \triangleleft y) & = & (dif.(p \triangleleft b).(q \triangleleft y) - dif.(a \triangleright p).(x \triangleright q))/(b - a). \end{array}$$

4. PList

The *PList* data structure was introduced in order to develop programs for the recursive problems which can be divided into any fixed number of subproblems. It is a generalization of *PowerList* data structure.

*PList*s are constructed with the n-way $|$ and $\natural$ operators; for the positive n, the n-way $|$ takes n similar *PList* and return their concatenation, and the n-way $\natural$ return their interleaving.

It will be used square brackets to denote ordered quantification in *PList* algebra. The expression $[|i : i \in \overline{n} : p.i]$ is a closed form for the application of the n-way operator $|$, applied to the *PList* $p.i$ in order. The range $i \in \overline{n}$ means that the terms of the expression are written from 0 trough $n - 1$ in the numeric order.

Formally, the *PList* constructors have the following types:

$$\begin{array}{lcl} < . > & : & X \to PList.X.1 \\ [|i : i \in \overline{n} : .] & : & (PList.X.n)^n \to PList.X.(n * m) \\ [\natural i : i \in \overline{n} : .] & : & (PList.X.n)^n \to PList.X.(n * m) \end{array}$$

where m is the length of the arguments, which are n similar *Plist*.

The *PList* axioms, also define the existence of unique decomposition of *PList* using constructors operators [1].

Functions over *PList* are defined using two arguments. The first argument is a list of arities: *PosList*, and the second is the *PList* argument. Functions over

PList are only defined for certain pairs of these input values; to express the valid pairs, it is require that the specification of the function defines the predicate:

$$defined : ((PosList \times PList) \to X) \times PosList \times PList \to Bool$$

to characterize where the function is defined.

We illustrate this, by defining the function *sum*. This function computes the sum of all elements of a *PList* over a type where + is defined:

$$\begin{array}{lcl} defined.sum.l.p & \equiv & prod.l = length.p \\ sum.[]. < a > & = & a \\ sum.(x \triangleright l).[|i : i \in x : p.i] & = & (+i : 0 \leq i < x : sum.l.(p.i)) \end{array}$$

where *prod.l* computes the product of the elements of list l, *length.p* is length of p, and [] denote the empty list.

Numerical Integration with Rectangle Formula

For a function $f : [a, b] \to \Re$, the integral $I = \int_a^b f.xdx$ can be approximate by [4]:

$$Q_{D_k}.f = \frac{1}{3} Q_{D_{k-1}}.f + h \sum_{i=1}^{2m} f.x_i,$$

where $h = \frac{b-a}{3^k}$, $m = 3^{k-1}$, $k = 1, 2, \ldots$ and the x_i values are computed by formulas:

$$\left\{ \begin{array}{lcl} x_1 & = & a + \frac{h}{2} \\ x_2 & = & a + \frac{5}{2}h \\ x_{2j+1} & = & x_1 + 2jh \\ x_{2j+2} & = & x_2 + 2jh, \ 1 \leq j < 3^{k-1}. \end{array} \right.$$

We want to define a *PList* function *drept*, that computes $(Q_{D_k}.f)$, for a fix k.

Let consider a division on interval $[a, b]$ with $n = 3^k$ points:

$$[x_0, \ldots, x_{n-1}] \quad = \quad [a_0, a_0 + \tfrac{h}{3^k}, \ldots, a_0 + \tfrac{3^k-1}{3^k}h], \text{ where } a_0 = a + \tfrac{h}{2}.$$

It can be notice that 3^{k-1} points are used for computation of $(Q_{D_{k-1}}.f)$ and $2 * 3^{k-1}$ intervene in computation of the second term, in the sum which computes $(Q_{D_k}.f)$.

The function

$$drept : Real \times PosList \times PList.Real.n \to Real$$

defined by:

$$\begin{array}{lcl} defined.sum.l.p & \equiv & prod.l = length.p \\ drept.[]. < x > & = & hk * x \\ drept.hk.(3 \triangleright l).[\natural i : i \in \overline{3} : p.i] & = & \frac{1}{3} * drept.(3 * hk).l.(p.1) + hk * sum.(2 \triangleright l).(p.0 \natural p.2), \end{array}$$

have three arguments; the first $hk = \frac{b-a}{3^k}$ is the division step, the second is a list form by k values equal with 3, and the third is the *PList* that contains the function values.

5. Fast Fourier Transformation

The *Discrete Fourier Transform* is an important tool used in many scientific applications. By this transformation, the polynomial representation with coeficients $(a_i, 0 \leq i < n)$ is changed to another. New representation consists of a set of values, which are the polynomial values in the nth order unity roots n, $(w_j, 0 \leq j < n)$. The polynomial degree n leads to the three cases.

A scalar function will be used in all the cases. The function $root : Nat \rightarrow Com$ applied to n returns the nth order unity root :

$$root.n = e^{\frac{2*\pi*\sqrt{-1}}{n}}.$$

5.1. **The case $n = 2^k$.** The formula that computes the polynomial value in w_j is:

$$f.w_j = \sum_{m=0}^{2^{k-1}-1} a_{2m} * e^{\frac{2\pi ijm}{2^{k-1}}} + e^{\frac{2\pi ij}{2^k}} \sum_{m=0}^{2^{k-1}-1} a_{2m+1} * e^{\frac{2\pi ijm}{2^{k-1}}},\ 0 \leq j < n.$$

It can be used *PowerList* data structure, for the parallel program specification, in this case. The function $fft : PowerList.Com.n \rightarrow PowerList.Com.n$ can be written as:

$$\begin{array}{lcl} fft. < a > & = & < a > \\ fft.(p \natural q) & = & (r + u * s) \mid (r - u * s) \end{array}$$

where

$$\begin{array}{lcl} r & = & fft.p \\ s & = & fft.q \\ u & = & powers.z.p \\ z & = & root.(length.(p \natural q)). \end{array}$$

The function $powers : Com \times PowerList.Com.n \rightarrow PowerList.Com.n$ is defined by:

$$\begin{array}{lcl} powers.x. < a > & = & < x^0 > \\ powers.x.(p \natural q) & = & powers.x^2.p \natural map.[x*].(powers.x^2.q)) \end{array}$$

where $[x*] : Com \rightarrow Com$ is the scalar function that multiplies its argument by x: $[x*].y = x * y$.

The function $powers.x.p$ returns a *PowerList* of the same length as p, containing the powers of x from 0 up to the length of p.

5.2. **The case n prime.** In this case, it is necessary to compute directly the polynomial values. The fft will have two *ParList* arguments, one formed by unity roots, and one formed by polynomial coeficients:

$$fft : ParList.Com.n \times ParList.Com.n \rightarrow ParList.Com.n$$

$$\begin{array}{lcl} fft. < z > .p & = & vp.z.p \\ fft.(u \mid v).p & = & fft.u.p \mid fft.v.p \\ fft.(z \triangleright u).p & = & vp.z.p \triangleright fft.u.p, \end{array}$$

where

$$vp : Com \times ParList.Com.n \to Com$$
$$vp.z.p = sum.(p * u), \qquad u = powers.z.p.$$

It has been used the extended operator $(*)$, for multiplies the *ParList* p and q. The function vp computes the polynomial value in one point and use the *powers* function, that is an extension of that presented in the first case, defined on *ParList*:

$$\begin{array}{lcl} powers.x. < a > & = & < x^0 > \\ powers.x.(p \natural q) & = & powers.x^2.p \natural map.[x*].(powers.x^2.q)) \\ powers.x.(a \triangleright q) & = & < x^0 > \triangleright map.[x*].(powers.x.q)). \end{array}$$

5.3. **The case** $n = r_1 * \cdots * r_k$**.** If n is not a power of two, but is a product of two numbers r_1 and r_2, the formula from the first case can be generalized in this way:

$$f.w_j = \sum_{k=0}^{r_1-1} \{ \sum_{t=0}^{r_2-1} a_{tr_1+k} e^{\frac{2\pi ijt}{r_2}} \} e^{\frac{2\pi ijk}{n}}, \quad 0 \le j < n.$$

The inner sum represents the value in $w_{j \ mod \ r_2}$, of the polynomial with degree equal with r_2 and the coeficients $\{a_k, a_{k+r_1}, \ldots, a_{k+r_1(r_2-1)}\}$. This value is computed by FFT for this polynomial. So, a recursive algorithm, that combines r_1 FFT, can be used.

[5] The best factorisation $n = r_1 * r_2$ for FFT (from the complexity point of view) is to chose r_1 from the prime factors of n.

Therefore, for the specification of the parallel program, we consider the decomposition in prime factors $n = r_1 * \cdots * r_k$. The *PList* data structures will be used. The arities list of the function is form by the prime factors of n.

$$fft : PosList \times PList.Com.n \to PList.Com.n$$

$$\begin{array}{lcl} defined.fft.l.p & \equiv & (prod.l = length.p) \\ fft.[x].[\natural i : i \in \overline{x} :< a.i >] & = & [|j : j \in \overline{x} : [+i : i \in \overline{x} : a.i * exp.z.(i * j)]] \\ fft.[x \triangleright l].[\natural i : i \in \overline{x} : p.i] & = & [|j : j \in \overline{x} : [+i : i \in \overline{x} : r.i * u.j]], \end{array}$$

where we used notations:

$$\begin{array}{lcl} r.i & = & fft.l.(p.i) \\ u.j & = & map.[exp.z.(j * \frac{n}{x})*].powers.z.l \\ z & = & root.n, \end{array}$$

and the functions:

$$exp : Com \times Nat \to Com$$
$$exp.x.i = x^i$$

$$powers : Com \times PosList \to PList.Com(prod.l)$$
$$powers.z.[] =< z^0 >$$
$$powers.z.(x \triangleright l) = [\natural i : i \in \overline{x} : map.[exp.z.i*].q]$$
where $q = powers.(exp.z.x).l.$

Some observations can be made:

- The base case represents the algorithm presented in the case n prime.
- If the list of arities contains, just values equal with 2, the program is maped to that in the case n is a power of two.

$$
\begin{aligned}
& [|j : j \in \overline{2} : [+i : i \in \overline{2} : r.i * u.j]] \\
= & \\
& (r.0 * u.0 + r.1 * u.0) \mid (r.0 * u.1 + r.1 * u.1) \\
= & \\
& (r.0 + r.1 * powers.z.l) \mid (r.0 + r.1 * [exp.z.\tfrac{n}{2}*].powers.z.l) \\
= & \\
& (r.0 + r.1 * powers.z.l) \mid (r.0 - r.1 * powers.z.l).
\end{aligned}
$$

6. Conclusions

The three data structures presented are useful in expressing parallel computations succinctly.

Using *PowerArray* theory presents some advantages such as: a layer of abstraction that is higher than that of the indices of elements, simplicity and correctness in developing parellel programs (Application 2).

The *ParList* theory provides an alternative extension of the *PowerList* theory to allow inputs of arbitrary lengths. This is done by using the *PowerArray* operators $\mid$ and $\natural$ to work with even lengths inputs and by using linear list operators $\triangleright$ and $\triangleleft$ to break a single element off an odd length list.

The *PList* notation is very rich. It includes the *PowerArray* theory as a special case. While this generality is not always needed in order to describe parallel computations, it may be useful when the problem is stated in a different radix than 2 (Application 4), or in a mixed radix as in the case of FFT with degree different from a power of two.

The correctness of the algorithms specified with these notations can be formally demonstrated, using the induction principles [1].

The three data structures can be extended to more than one dimension by replicating the constructors for each dimension. This allows to describe matrix computations, using a similar approach.

The examples presented illustrate the power of these notations.

References

[1] Jacob Kornerup, *Data Structures for Parallel Recursion*, Ph thesis, University of Texas at Austin, 1997.

[2] Jacob Kornerup, *PLists: Taking PowerLists Beyond Base Two*, First International Workshop on Constructive Methods for Parallel Programming, MIP-9805, May 1998.

[3] Jacob Kornerup, *Mapping a functional notation for parallel programs onto hypercubes*, Information Processing Letters, 53:153-158, 1995.

[4] Gh. Coman, *Numerical Analysis*, Editura Libris, Cluj-Napoca, 1995 (in Romanian).

[5] H.S. Wilf, *Algoritmes et complexite*, Mason & Prentice Hall,1985.

"Babeş-Bolyai" University, Department of Computer Science, Cluj-Napoca, RO 3400
E-mail address: gina@cs.ubbcluj.ro